React State, Props & Hooks

What are React Hooks?

- A way to use state and side-effects in React function components.
- A Hook is a special function that lets you “hook into” React features.

WHY REACT HOOKS?

- It’s hard to reuse stateful logic between components. Patterns like **render props** and **higher-order** components that try to solve this.
- Complex components become hard to understand.
- Classes confuse both people and machines.

State Hook

```
class Example extends React.Component {
   constructor(props) {
     super(props);
     this.state = {
       count: 0
     };
   }
    render() {
     return (
       <div>
         <p>You clicked {this.state.count} times</p>
         <button onClick={() => this.setState({ count: this.state.count + 1 })}>
           Click me
         </button>
       </div>
     );
   }
 }
```

```
import React, { useState } from 'react';

function Example() {
 // Declare a new state variable, which we'll call "count"
 const [count, setCount] = useState(0);

 return (
   <div>
     <p>You clicked {count} times</p>
     <button onClick={() => setCount(count + 1)}>
       Click me
     </button>
   </div>
 );
}
```

State Hook

```
const [state, setState] = useState(initialState);
```

- Returns a stateful value, and a function to update it.
- During the initial render, the returned state (state) is the same as the value passed as the first argument (initialState).
- The setState function is used to update the state. It accepts a new state value and enqueues a re-render of the component.
- During subsequent re-renders, the first value returned by useState will always be the most recent state after applying updates.

Effect Hook

```
class Example extends React.Component {
   constructor(props) {
     super(props);
     this.state = {
       count: 0
     };
   }
    componentDidMount() {
     document.title = `You clicked ${this.state.count} times`;
   }
   componentDidUpdate() {
     document.title = `You clicked ${this.state.count} times`;
   }
    render() {
     return (
       <div>
         <p>You clicked {this.state.count} times</p>
         <button onClick={() => this.setState({ count: this.state.count + 1 })}>
           Click me
         </button>
       </div>
     );
   }
 }
```

```
import React, { useState, useEffect } from 'react';

function Example() {
 const [count, setCount] = useState(0);

 // Similar to componentDidMount and componentDidUpdate:
 useEffect(() => {
   // Update the document title using the browser API
   document.title = `You clicked ${count} times`;
 });

 return (
   <div>
     <p>You clicked {count} times</p>
     <button onClick={() => setCount(count + 1)}>
       Click me
     </button>
   </div>
 );
}
```

Effect Hook

```
useEffect(didUpdate);
```

- By default, effects run after every completed render, but you can choose to fire them only when certain values have changed.
- The function passed to useEffect may return a clean-up function.
- If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument.

Rules of Hooks

1. Only Call Hooks at the Top Level

- Don’t call Hooks **inside loops**, **conditions**, or **nested functions**.
- Use Hooks at the top level of your React function.

2. Only Call Hooks from React Functions

- Don’t call Hooks from regular JavaScript functions.
- Call Hooks from React function components.
- Call Hooks from custom Hooks.

Other Hooks

Basic Hooks

- useState
- useEffect
- useContext

Additional Hooks

- useReducer
- useCallback
- useMemo
- useRef

React Destructuring

React – What is Destructuring?

Destructuring is a simple property that is used to make code much clear and readable, mainly when we pass props in React.

React – What is Destructuring?

- **Destructuring** is a characteristic of JavaScript, It is used to take out sections of data from an array or objects, We can assign them to new own variables created by the developer.

- In **destructuring**, It does not change an array or any object**, it makes a copy of the desired object or array element by assigning them in its own new variables**, later we can use this new variable in React (class or functional) components.

React – What is Destructuring?

- It makes the code more clear. When we access the props using this keyword, we have to use **this/ this.props** throughout the program, but by the use of restructuring, we can discard **this/ this.props** by assigning them in new variables.

- This is very difficult to monitor **props** in complex applications, so by assigning these props in new own variables we can make a code more readable.

React – Destructuring Advantages

- It makes developer’s life easy, by assigning their own variables.
- Nested data is more complex, it takes time to access, but by the use of destructuring, we can access faster of nested data.
- It improves the **sustainability**, **readability** of code.
- It helps to cut the amount of code used in an application.

React – Destructuring Advantages

- It trims the number of steps taken to access data properties.
- It provides components with the exact data properties.
- It saves time from iterate over an array of objects multiple times.
- In ReactJS We use multiple times ternary operators inside the render function, without destructuring it looks complex and hard to access them, but by the use of destructuring, we can improve the readability of ternary operators.

React – Destructuring Advantages

- **2. Using the Extraction method:**

There are many times when the value extracted during Destructuring no more exist, then in this condition we can use of default behavior of Destructuring, in this, apply a default value to the newly declared properties of Destructuring

In the following code, the **activeObject** will be set **true** if it is undefined in *this.props*.

```
Const {active, activeStatus, activeObject = true } = this.props
```

React – Destructuring Advantages

- **3. Using the Re-assigning method:**

A variable name that is not a copy of the property being destructured may be used. This is achieved by reassigning as shown below.

In the following code, the properties active, activeStatus have been destructured and reassigned as variables named generating, objectMessage.

```
const { active : generating, activeStatus : objectMessage} = this.props
```

Further reading

Further reading

- https://reactjs.org/docs/hooks-intro.html
- https://levelup.gitconnected.com/learn-react-usestate-hook-a09ccf955537
- https://www.w3schools.com/react/react_usestate.asp

Quiz

Quiz

- https://www.w3schools.com/quiztest/quiztest.asp?qtest=REACT
- https://create.kahoot.it/details/27db8904-3d09-40ee-ace2-3ae4483cc3b0

Assignment

Assignment – 1 (exercitiile de la 1 – 8)

- https://coderfiles.dev/blog/reactjs-coding-exercises/
- To be commited and pushed in folder e.g. C15/assignment/exercise1 (and so on 1, 2, 3) in your branch
- Transform in callback functions the inline functions from components

Assignment extra – from state to reducer

- https://www.robinwieruch.de/react-state-usereducer-usestate-usecontext

You have the code here:

https://codesandbox.io/s/1jvy0?file=/src/index.js

Documentation on the left